

High Performance Computing - HW 6

Anthony Maylath

May 12, 2019

The Courant Institute for Mathematical Sciences, New York University

1 Question 0

We have completed an MPI implementation of the Kmeans and EM algorithms. At this point, Kmeans is pretty optimized while we are still improving the EM implementation. We ran scalability and timing tests on Kmeans and found descent performance. Our plan is to polish our EM implementation using similar HPC techniques and compare performance and results between the two. We have not yet started compiling the report or presentation slides.

2 Question 1

My implementation can be found *jacobi_mpi.cpp*. The implementation takes two command line arguments: the first sets the dimension of the problem, N , and the second specifies the max number of Jacobi iterations. My implementation uses row major ordering for the solution lu . I send a different communication depending on where the process falls on the grid. If the process computes the solution on the top row, then don't communicate ghost values for the top. I filter out top row communication by checking if $mpirank \geq \sqrt{p}$ where p is the number of processes. I apply similar checks on all four edges of ghost values.

Figures 1 and 2 demonstrate the weak scaling of the implementation for 64 nodes with 4 cores per node and 16 nodes with 16 cores per node respectively. Each line represents a fixed number of Jacobi iterations. The base size of the problem is $N = 100$ and is scaled up proportional to the number of processes so that each process gets $lN = 100$ locally. The secondary axis corresponds to 1,000 and 100 iterations. For 100 iterations, it seems like we lose some performance for larger problem sizes, while the performance time seems more consistent across problem sizes for more iterations.

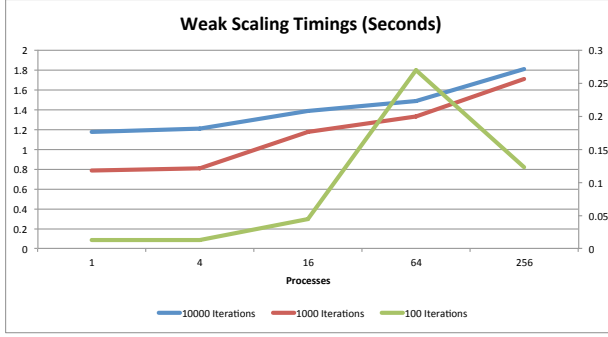


Figure 1: Weak Scaling 64 Nodes, 4 Cores Per Node

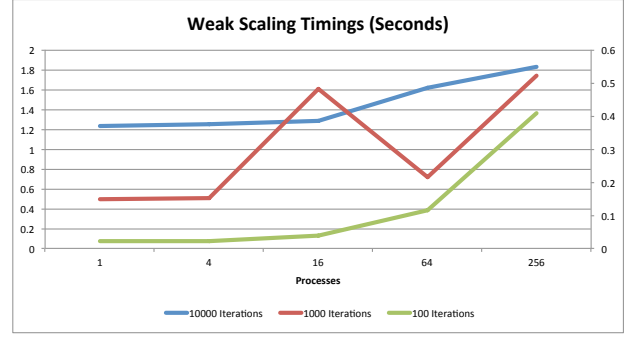


Figure 2: Weak Scaling 16 Nodes, 16 Cores Per Node

3 Question 2

sample_sort.cpp contains my implementation of sample sort with mpi processes. I start by assigning each process a list of integers of size $lN = N/p$ where N is specified as the first command line argument. The local lists are initialized with the *rand()* function in the standard library. The splitter candidates are selected to be the first $p - 1$ elements of each local list, where p is the total number of processes. Since the numbers are randomly initialized, these first elements will have random order. To perform local sorting, I use quicksort. The implementation is located in header file, *hw6helper*, and is a modification of a quicksort algorithm from stack overflow. The sorted output is stored in files named *output(rank).txt* where rank is the local mpi rank. Each file has elements in ascending order and files with larger ranks contain larger elements.

I ran my timings on Prince with 16 nodes, 16 tasks per node and 16GB of memory. Figures 3 and 4 graph the strong and weak scaling respectively. Each symbol in 3 represents a list size. We see larger list sizes give better strong scaling. Also, the performance seems to go down if we use too many processes. The optimal processes seems to be 16 and 50 for list size 1 million and 10 million respectively. Perhaps we need a larger list size to see optimal performance with over 100 processes. Figure 4 keeps the local list size as 10,000. The global size is increased with the number of processes. The run time goes up as we increase the problem size, except when we move from 4 to 10 processes when it goes down. The trend line indicates a slope of 0.0148 which implies that about 0.02 seconds are added to the run time when the list size is increased by 10,000 and the number of processes are increased by 1. The R^2 of the trendline is quite high at 0.998 which suggests the weak scaling really is linear; however, we probably need more datapoints to make the result statistically significant.

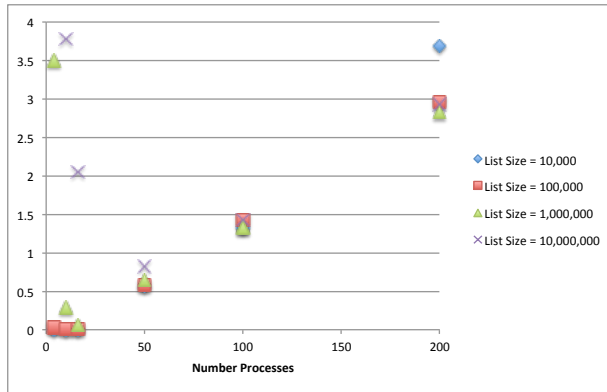


Figure 3: Sample Sort: Strong Scaling Timings

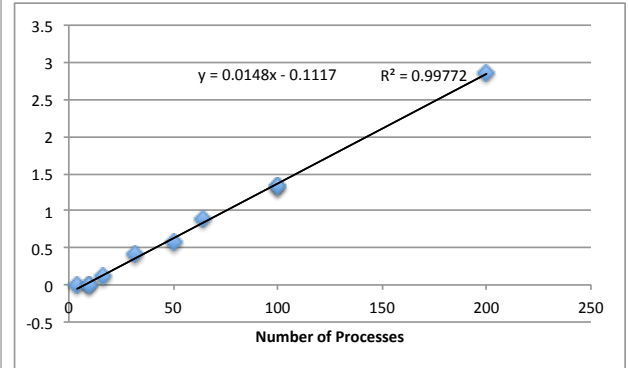


Figure 4: Weak Scaling for 10,000 Points per Process - Timings

4 Question 3: Extra Credit

I opted to parallize the one dimensional multigrid method in openmp. The my implementation is an alteration of the code available on git for lecture13. The implementation now requires 4 command line parameters with the fourth parameter being the number of omp threads. I added parallel for statements to the `set_zero`, `compute_residual`, and `jacobi` functions. In addition, I saw speed up by adding parallel for statements to the `coarsen` and `refine_and_add` functions despite an exponentially decreasing amount of work with each `v`-step. These functions also have parallel for pragmas in my implementation. Instead of performing a `memcpy` with each `jacobi` step, I simply switch the `u` and `unew` pointers. This speeds things up, but requires the `ssteps` parameter to be even. Finally, I added a reduction to the function `compute_norm`.

I attempted to run my implementation on Prince, but I did not see any speed up. For `omp_threads = 1,...,10` the run time was always around 10 seconds for $N = 100$ million. However, I was able to get a clear speedup on CIMS crackle2. Figures 5 and 6 show the strong and weak scaling respectively for `omp_threads = 1, 2, ..., 10`. For strong scaling, I always use 700 million points. For weak scaling I start with 100 million points and add another 100 million points for each additional omp thread. For strong scaling, the optimal number of omp threads for 700 million points seems to be between 4 and 7. For weak scaling, a line fit estimates that each omp thread adds about 6 seconds of run time. The weak scaling isn't great but its much better than running serially as the serial code takes about 9 seconds for 100 million points. If we scale this up linearly, that implies the serial code should take around 90 seconds for 1 billion points. However, we see better performance on the parallel implemenation with around 63 seconds on 1 billion points and 10 threads.

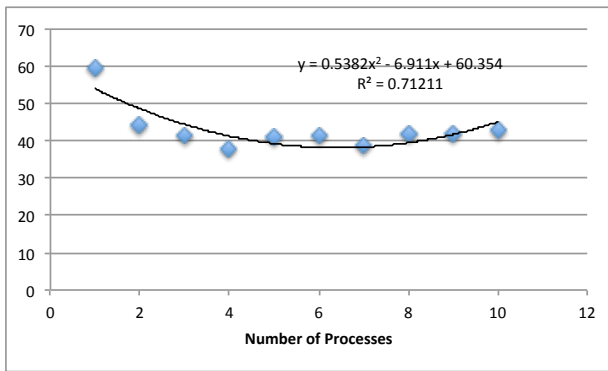


Figure 5: crackle2 - Strong Scaling Timings

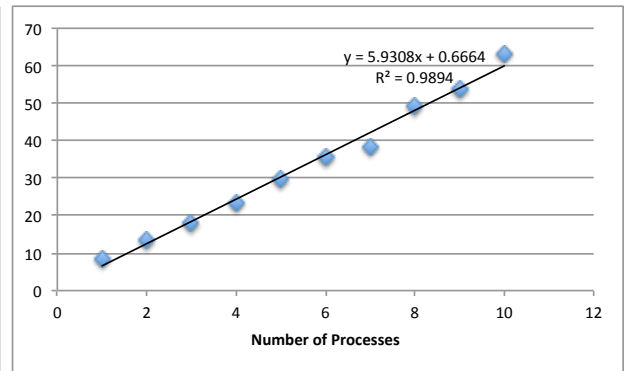


Figure 6: crackle2 - Weak Scaling Timings